

Understanding the OCFA Abstraction

Kristopher Micinski

CIS 700 — Program Analysis: Foundations and Applications
Fall '19, Syracuse University



AAM is a **general** strategy for building abstract interpreters from abstract machines

But what do the results **mean**?

For the first part of this lecture—let's just figure out the answer intuitively (without implementing the analysis)

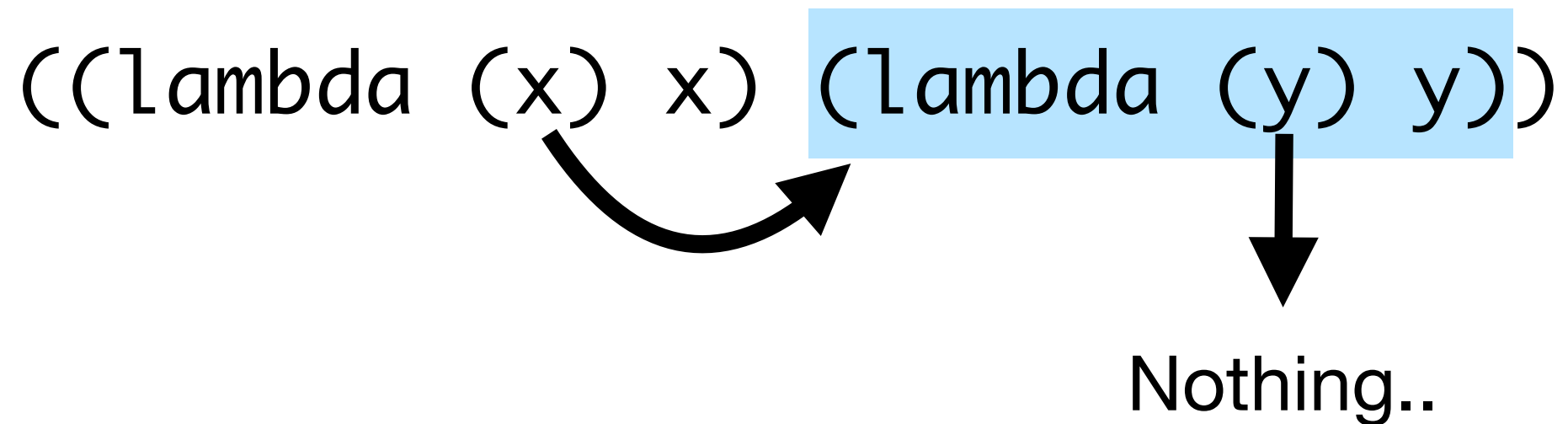
This week we will study **finite** analyses

Today, will study 0CFA—finite flow analysis for Scheme

On Wednesday, k-CFA—Improves precision of 0CFA

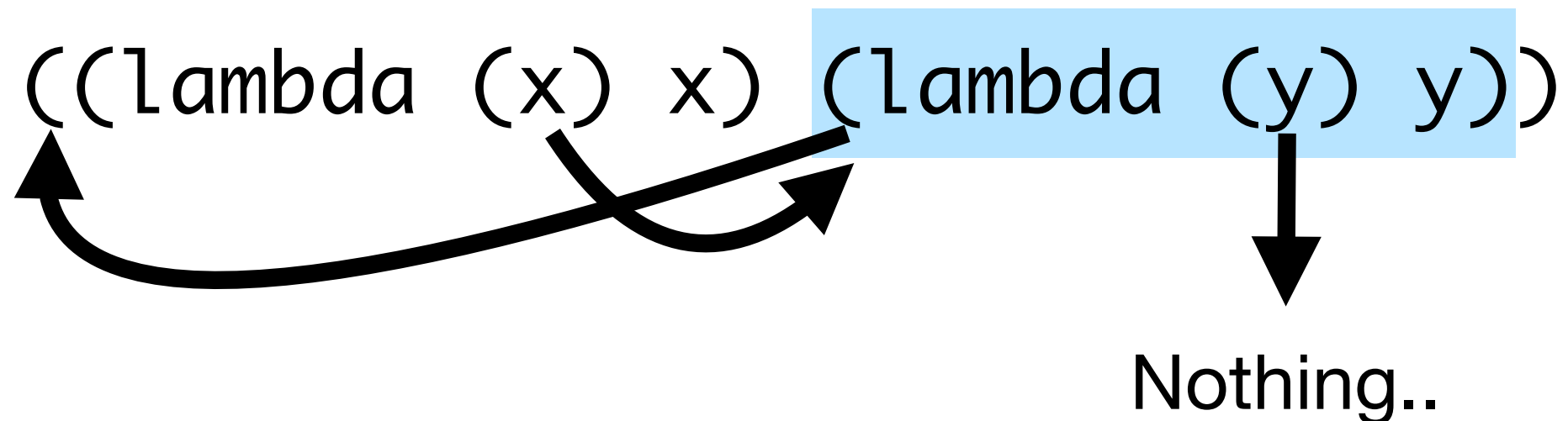
OCFA

Asks the question:
for each **variable**, which possible **lambdas**
could be **bound** to that variable?



OCFA

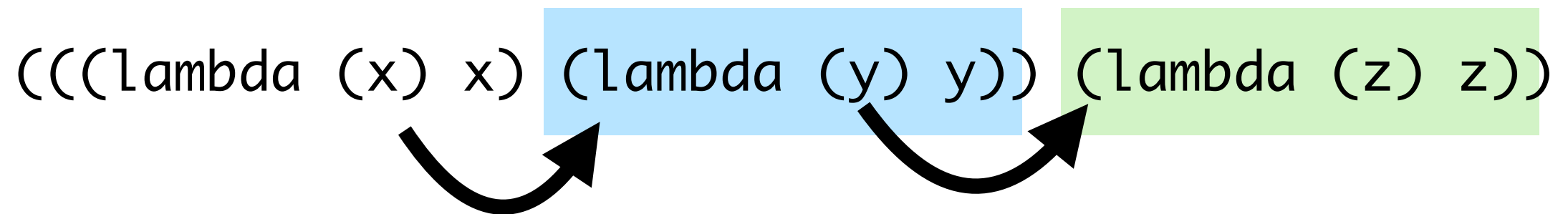
Asks the question:
for each **variable**, which possible **lambdas**
could be **bound** to that variable?



Also...

`(lambda (y) y)` flows to result of **entire application**...

The issue is that this could require transitive reasoning...



Rules for OCFA

For the lambda calculus...

$$e ::= (\text{lambda } (x) \ e) \\ \quad | \ (e \ e) \\ \quad | \ x$$

- For each lambda term, $(\text{lambda } (x) \ e)$:
 - $(\text{lambda } (x) \ e)$ flows to itself
- For each application $(e_0 \ e_1)$...
 - If $(\text{lambda } (x) \ e')$ flows to e_0 and v flows to e_1 ...
 - Then v flows to x
- For each application $(e_0 \ e_1)$:
 - If $(\text{lambda } (x) \ e)$ flows to e_0 and v flows to body e ...
 - Then v flows to $(e_0 \ e_1)$

Rules for OCFA

For the lambda calculus...

$$e ::= (\text{lambda } (x) \ e) \\ \quad | \ (e \ e) \\ \quad | \ x$$

- For each lambda term, (lambda (x) e): **Base Case**
 - (lambda (x) e) flows to itself
- For each application (e0 e1)... **Calls**
 - If (lambda (x) e') flows to e0 and v flows to e1...
 - Then v flows to x
- For each application (e0 e1): **Returns**
 - If (lambda (x) e) flows to e0 and v flows to body e...
 - Then v flows to (e0 e1)

- For each lambda term, $(\text{lambda } (x) \ e)$:
 - $(\text{lambda } (x) \ e)$ flows to itself
- For each application $(e_0 \ e_1)$...
 - If $(\text{lambda } (x) \ e')$ flows to e_0 and v flows to e_1 ...
 - Then v flows to x
- For each application $(e_0 \ e_1)$:
 - If $(\text{lambda } (x) \ e)$ flows to e_0 and v flows to body e ...
 - Then v flows to $(e_0 \ e_1)$

OCFA assigns a set of expressions to each bound variable **and** each subexpression in the program

Intuitively, “what flows to each variable and returns to each control point.”

$((\text{lambda } (x) x) (\text{lambda } (y) y))$

- For each lambda term, $(\text{lambda } (x) e)$:
 - $(\text{lambda } (x) e)$ flows to itself
- For each application $(e_0 e_1)$...
 - If $(\text{lambda } (x) e')$ flows to e_0 and v flows to e_1 ...
 - Then v flows to x
- For each application $(e_0 e_1)$:
 - If $(\text{lambda } (x) e)$ flows to e_0 and v flows to body e ...
 - Then v flows to $(e_0 e_1)$
- $(\text{lambda } (x) x)$ flows to itself, $(\text{lambda } (y) y)$ flows to itself
- For the application:
 - $(\text{lambda } (x) x)$ flows to $(\text{lambda } (x) x)$ and...
 - $(\text{lambda } (y) y)$ flows to $(\text{lambda } (y) y)$
 - So $(\text{lambda } (y) y)$ flows to x
- (Now, the third rule...)
 - We just decided that $(\text{lambda } (y) y)$ flows to x
 - Thus, $(\text{lambda } (y) y)$ also flows to the entire application!

$(((\text{lambda } (x) \ x) (\text{lambda } (y) \ y)) (\text{lambda } (z) \ z))$

- For each lambda term, $(\text{lambda } (x) \ e)$:
 - $(\text{lambda } (x) \ e)$ flows to itself
- For each application $(e_0 \ e_1)$...
 - If $(\text{lambda } (x) \ e')$ flows to e_0 and v flows to e_1 ...
 - Then v flows to x
- For each application $(e_0 \ e_1)$:
 - If $(\text{lambda } (x) \ e)$ flows to e_0 and v flows to body e ...
 - Then v flows to $(e_0 \ e_1)$

Work through this one on board...

Now try Ω ...

- For each lambda term, $(\text{lambda } (x) \ e)$:
 - $(\text{lambda } (x) \ e)$ flows to itself
- For each application $(e_0 \ e_1)$...
 - If $(\text{lambda } (x) \ e')$ flows to e_0 and v flows to e_1 ...
 - Then v flows to x
- For each application $(e_0 \ e_1)$:
 - If $(\text{lambda } (x) \ e)$ flows to e_0 and v flows to body e ...
 - Then v flows to $(e_0 \ e_1)$

$((\text{lambda } (x) \ (x \ x)) \ (\text{lambda } (y) \ (y \ y)))$

We can intuitively extend the lambda calculus with various constructs fairly easily...

`((lambda (x) x)
 (if #t (lambda (y) y) (lambda (z) z))))`

The diagram illustrates data flow in a lambda calculus expression. A curved arrow originates from the lambda expression `(lambda (x) x)` and branches into two straight arrows. One arrow points to the blue-highlighted lambda expression `(lambda (y) y)` within the `(if #t ...)` branch, and the other points to the green-highlighted lambda expression `(lambda (z) z)` within the same branch. This indicates that the function `(lambda (x) x)` is passed to both possible execution paths of the conditional.

`((lambda (x) x)
 (if #t (lambda (y) y) (lambda (z) z))
 (lambda (a) a)))`

This code snippet shows a similar structure but with an additional branch. The `(if #t ...)` branch contains two lambda expressions: `(lambda (y) y)` (highlighted in blue) and `(lambda (z) z)` (highlighted in green). A third branch, `(lambda (a) a)` (highlighted in red), follows the `(if ...)` construct. This represents a function `(lambda (x) x)` being passed to a conditional that chooses between two functions, or a default function `(lambda (a) a)` if the condition is false.

Practice: what flows where?

Data flow depends on **control** flow

`((lambda (x) x)`
`(if #t (lambda (y) y) (lambda (z) z)))`

The diagram illustrates control flow. A curved arrow originates from the `(lambda (x) x)` expression and branches into two arrows. One arrow points to the `(lambda (y) y)` expression, which is highlighted with a light blue background. The other arrow points to the `(lambda (z) z)` expression, which is highlighted with a light green background.

`((lambda (x) x)`
`(if #t (lambda (y) y) (lambda (z) z)))`
`(lambda (a) a))`

The diagram illustrates control flow. A curved arrow originates from the `(lambda (x) x)` expression and branches into two arrows. One arrow points to the `(lambda (y) y)` expression, which is highlighted with a light blue background. The other arrow points to the `(lambda (z) z)` expression, which is highlighted with a light green background. A third curved arrow originates from the `(lambda (z) z)` expression and points to the `(lambda (a) a)` expression, which is highlighted with a light red background.

Note: also need to know what expressions
flow to which control points

`((lambda (x) x)
 (if #t (lambda (y) y) (lambda (z) z))))`

The diagram illustrates the flow of a lambda expression `(lambda (x) x)` to the condition `#t` and the 'then' branch `(lambda (y) y)` of an `if` statement. The `(lambda (y) y)` branch is highlighted in blue, and the `(lambda (z) z)` branch is highlighted in green.

`((((lambda (x) x)
 (if #t (lambda (y) y) (lambda (z) z)))
 (lambda (a) a)))`

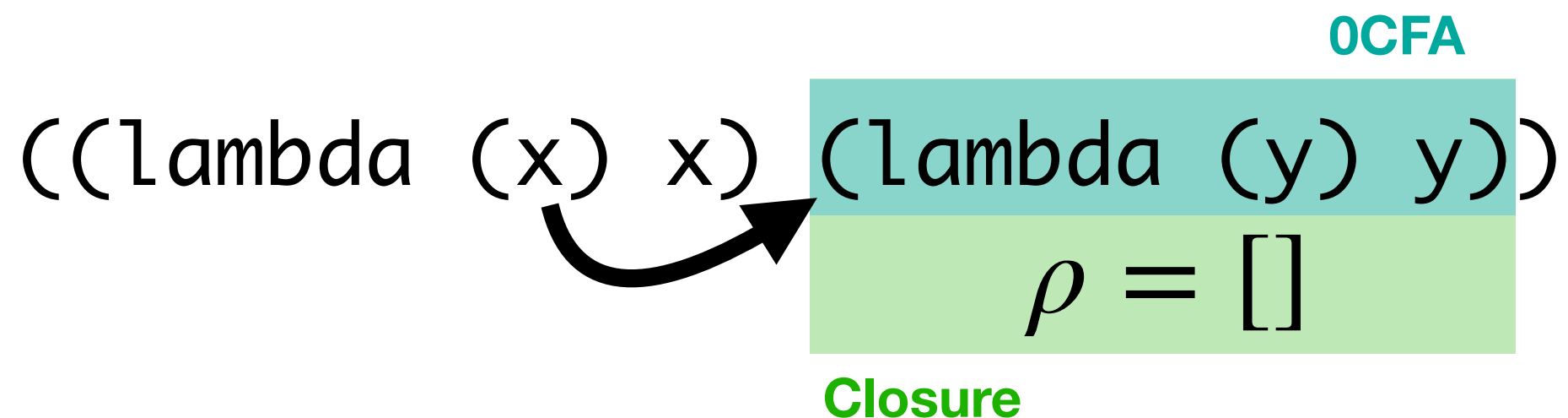
The diagram illustrates the flow of a lambda expression `((lambda (x) x) ...)` to the condition `#t`, the 'then' branch `(lambda (y) y)`, and the 'else' branch `(lambda (a) a)` of an `if` statement. The `(lambda (y) y)` branch is highlighted in blue, the `(lambda (z) z)` branch is highlighted in green, and the `(lambda (a) a)` branch is highlighted in red.

One choice: anything that flows to either side of an if
could return from the entire if
(Can refine via adding precision, will do this next.)

This is an approximation, because at runtime **lambdas** don't get bound to variables, but **closures** do.

OCFA conflates all possible environments that could be closed alongside a piece of syntax

OCFA “ignores” the environment

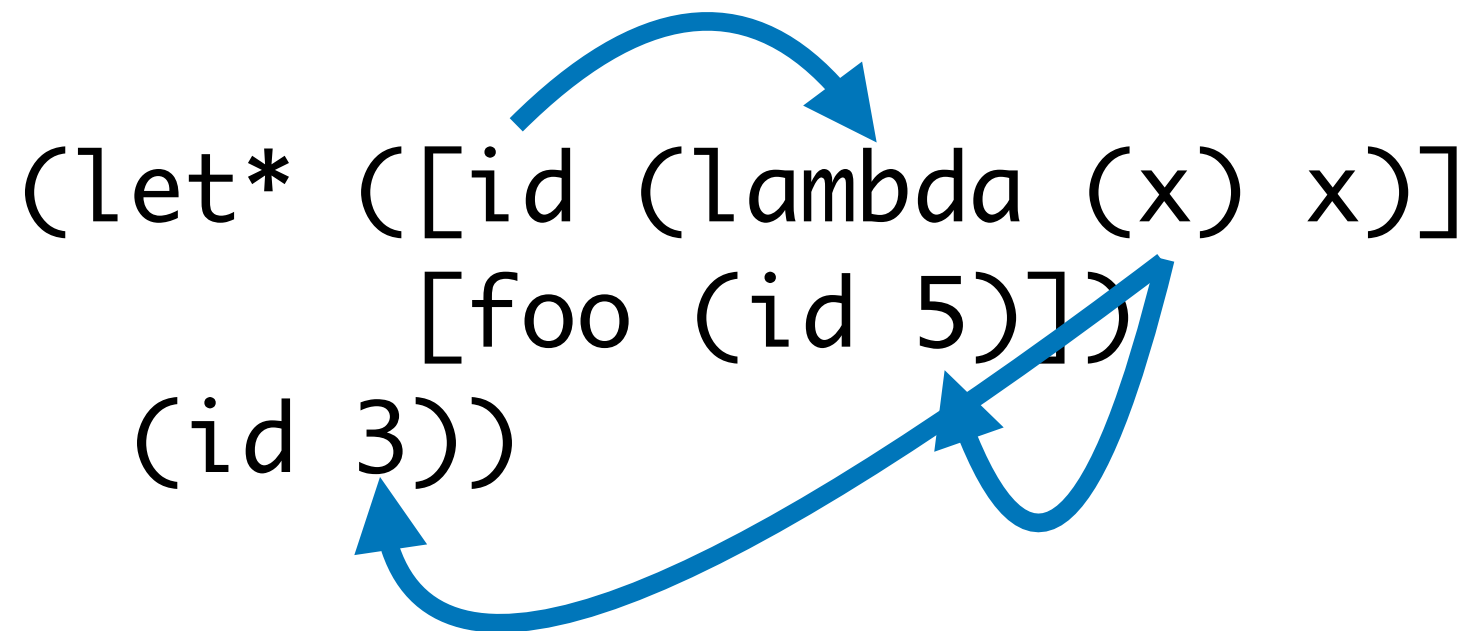


Can treat let/let* similarly to lambdas...

```
(let* ([id (lambda (x) x)]  
      [foo (id 5)])  
  (id 3))
```

What does OCFA say x will get bound to..?

Can treat let/let* similarly to lambdas...



Subsequently, OCFA says that any return from `id` also returns to any callsite for `id`

Note: this illustrates ambiguity in OCFA!

Thinking back to last time...

$$\hat{\varsigma} \longmapsto_{\widehat{CESK}_t^*} \hat{\varsigma}', \text{ where } \kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(t, \kappa)$$

$\langle x, \rho, \hat{\sigma}, a, t \rangle$	$\langle v, \rho', \hat{\sigma}, a, u \rangle$ where $(v, \rho') \in \hat{\sigma}(\rho(x))$
$\langle (e_0 e_1), \rho, \hat{\sigma}, a, t \rangle$	$\langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u \rangle$
$\langle v, \rho, \hat{\sigma}, a, t \rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(v, \rho, c)], b, u \rangle$
if $\kappa = \mathbf{fn}((\lambda x.e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto (v, \rho)], c, u \rangle$

Figure 5. The abstract time-stamped CESK^{*} machine.

[Van Horn and Might, '10]

Defines a **family** of interpreters
(Instantiate by choosing alloc / tick appropriately.)

$$\hat{\varsigma} \longmapsto_{\widehat{CESK}_t^*} \hat{\varsigma}', \text{ where } \kappa \in \hat{\sigma}(a), b = \widehat{alloc}(\hat{\varsigma}, \kappa), u = \widehat{tick}(t, \kappa)$$

$\langle x, \rho, \hat{\sigma}, a, t \rangle$	$\langle v, \rho', \hat{\sigma}, a, u \rangle$ where $(v, \rho') \in \hat{\sigma}(\rho(x))$
$\langle (e_0 e_1), \rho, \hat{\sigma}, a, t \rangle$	$\langle e_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathbf{ar}(e_1, \rho, a)], b, u \rangle$
$\langle v, \rho, \hat{\sigma}, a, t \rangle$	
if $\kappa = \mathbf{ar}(e, \rho', c)$	$\langle e, \rho', \hat{\sigma} \sqcup [b \mapsto \mathbf{fn}(v, \rho, c)], b, u \rangle$
if $\kappa = \mathbf{fn}((\lambda x.e), \rho', c)$	$\langle e, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto (v, \rho)], c, u \rangle$

Figure 5. The abstract time-stamped CESK^{*} machine.

[Van Horn and Might, '10]